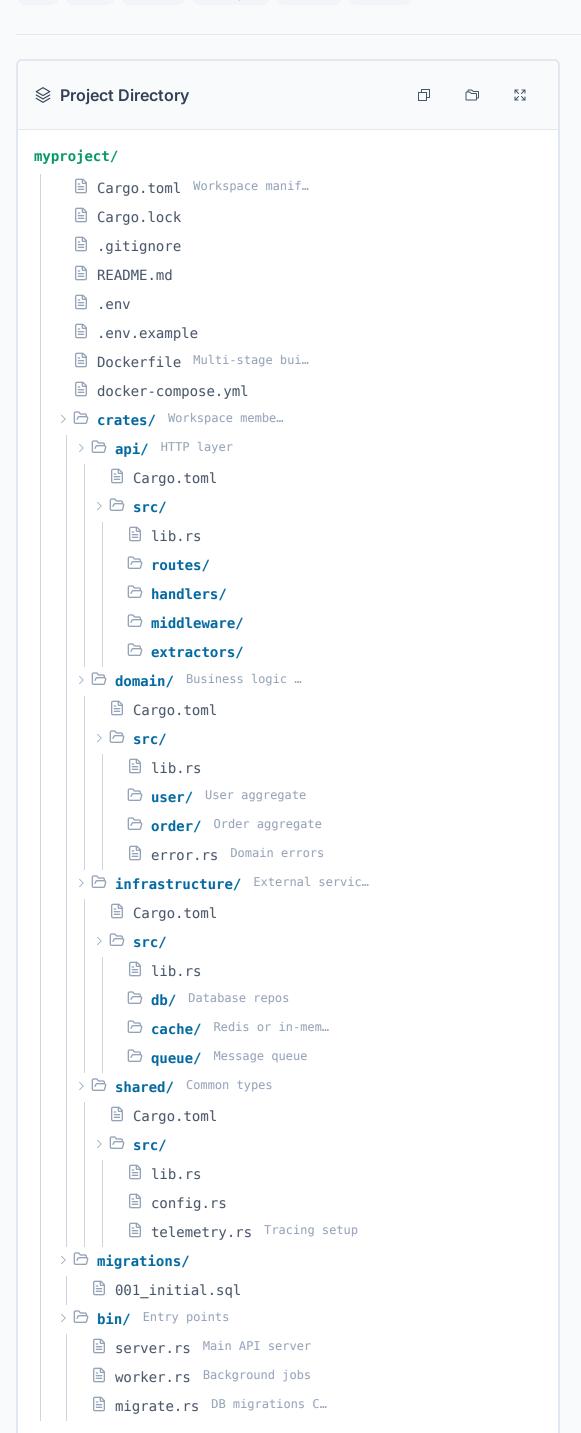
# **R** Axum Modular Project Structure

Multi-crate workspace for large projects. Separate crates for domain, API, and infrastructure.

#rust #axum #modular #workspace #backend #scalable





## **○** Why This Structure?

Cargo workspaces let you split a project into multiple crates with shared dependencies. The domain crate has zero external dependencies—pure Rust types and logic. The api crate handles HTTP, infrastructure implements storage. This enforces boundaries at compile time.

## 

crates/api/ - Axum server, routes, handlers, middleware
crates/domain/ - Pure business logic, no framework deps
crates/infrastructure/ - Database, cache, queue implementations
crates/shared/ - Config, telemetry, common utilities
bin/ - Binary entry points for different services

## </> Workspace Configuration

```
# Cargo.toml (workspace root)
[workspace]
members = ["crates/*"]
resolver = "2"

[workspace.dependencies]
axum = "0.7"
tokio = { version = "1", features = ["full"] }
sqlx = { version = "0.8", features = ["postgres"] }
```

## Getting Started

- 1. mkdir myproject && cd myproject
- 2. Create workspace Cargo.toml
- 3. cargo new crates/api --lib
- 4. cargo new crates/domain --lib
- 5. cargo run --bin server

## ☑ When To Use This

- Large codebases with multiple domains
- Team of 5+ developers
- Need compile-time boundary enforcement
- Multiple binaries (server, worker, CLI)
- Long-lived projects (2+ years)

## 

Complex setup - Workspace config and crate dependencies take time
More ceremony - Cross-crate imports require careful planning
Slower iteration - Changes may rebuild multiple crates
Overkill for small projects - Single-crate is faster for prototypes

## **☑** Best Practices

- Domain crate should never import api or infrastructure
- Use traits in domain, implement in infrastructure
- Shared crate for cross-cutting concerns only
- Keep bin/ files minimal—just wire and run
- Use workspace.dependencies for version consistency